
# Tivoli Directory Integrator v7 exercises

***Version 1.3***

***September 2011***

***Skills 1st Ltd***

***www.skills-1st.co.uk***

Dr Andrew Findlay
Skills 1st Ltd
2 Cedar Chase
Taplow
Maidenhead
SL6 0EU
01628 782565
andrew.findlay@skills-1st.co.uk

# Table of Contents


1 Working Environment....3
1.1 Preparation....3
2 Installing TDI....4
3 The Config Editor....6
4 A simple assembly line....7
5 TDI Debugger....9
6 Attribute Maps....10
6.1 Using the output map....10
6.2 In-line attribute maps....10
7 Flow Control....12
7.1 Using IF....12
7.2 ELSE....12
8 Database Lookup....13
8.1 Installing the MySQL JDBC connector....13
8.2 Configuring the TDI connector....13
9 Hooks and Scripts....15
9.1 Preparation....15
9.2 Dealing with missing data....15
9.3 Multiple hits....16
9.4 Script objects and methods....16
10 Connector Loops....17
11 Adding and Updating LDAP entries....19
11.1 Building the LDAP Server....19
11.2 Installing Apache Directory Studio....19
11.3 Connecting TDI to LDAP....20
11.4 Summary....22
12 Property Stores....23
13 Handling multiple-hit results....25
14 Server-Mode connectors....26
15 Delta-Mode connectors....27
15.1 Configure the System Store....27
15.2 Connect the CE to the System Store....27
15.3 Simple Delta Mode....27
15.4 Using Delta Data....28
16 TLS and X.509 certificates....29
16.1 TLS in LDAP....29
16.2 A new assembly line....29
16.3 Negotiating TLS....30
16.4 Key and Certificate management....30
16.5 A malicious LDAP server....31
17 Preparing for deployment....33
18 AMC....34
19 Data Cleanup and Reporting....35
20 LDAP to SQL synchronisation....36
21 Mailing List from LDAP Group....37

22 Generating unique IDs safely using LDAP.....38
23 Account Provisioning using LDAP.....39

# 1 Working Environment

All the exercises will be done in a VMware virtual machine running SuSE Linux (SLES 10).

Login as *student* password *skills782565*

For a few of the installation steps you will need to work as *root*. It is best practice to avoid this where possible. The password is also *skills782565*

The virtual machine includes a MySQL database. For admin purposes you may need to use the MySQL *root* user, whose password is *example*

A number of LDAP server configurations are provided as part of the course kit. All of these run under the *student* username. Their management DNs and passwords are provided in files called `vars` in each server directory, but in most cases you will be connecting using an account provided specifically for TDI. The LDAP servers are in subdirectories of `~/ldap` and a set of scripts is provided to manipulate them. The scripts are in `~/ldap/bin` which is already in your PATH, and all have names starting "`x-`"

Data files for exercises can be found in `~/files`

Documentation for TDI and for JavaScript can be found in folders on the desktop.

## 1.1 Preparation

Download TDI 7.1 from IBM. You need the 32-bit Intel-Linux Identity Edition. The free time-limited evaluation version is OK: it will have a filename of the form `tdiv71_ie_trial_linux_x86.tar` and it should be placed in `~student/files`

Similarly, download the latest fixpack and put it in the same directory.

# 2 Installing TDI

The TDI install kit is in the file `tdiv71_ie_trial_linux_x86.tar`

It is possible to install TDI as a non-root user, but for this class we will use root so that TDI goes into the default location.

1. Login as *student*, password *skills78256*
2. Open a shell window
3. Make a new directory and unpack the installation kit in it:

```
cd
mkdir tdi-kit
cd tdi-kit
tar -x -p -f ~/files/tdiv71_ie_trial_linux_x86.tar
```

4. We will bypass the web-based 'launchpad' and go directly to the installer:

```
cd linux_x86
```

5. At this point you need to become root to run the installer. The root password is *skills78256*

```
su
./install_tdiv71_linux_x86.bin
```

6. The GUI installer will start. Unfortunately this version of TDI has no native package install method (e.g. Linux RPMs). If you need to install it on a machine with no graphics, you can run the installer in text mode or you may be able to copy `/opt/IBM/TDI` from another machine.
7. Follow through the installation wizard. Accept the default install directory: `/opt/IBM/TDI/V7.1`
8. Select the Custom Installation option and look at the list of components available. Accept the six that are ticked by default.
9. For the Solutions Directory, select 'Use a directory named TDI under my home directory'. There is a bug in the way this is implemented which we will correct shortly.
10. Make a note of the port numbers chosen for the TDI Server – probably Server:1099 System Store:1527 REST API:1098 and System Queue:41001.
11. Choose to register TDI as a system service.
12. Accept the (default) Embedded Instance of the Integrated Solutions Console.
13. Make a note of the port numbers chosen for the Integrated Solutions Console – probably HTTP:13100 HTTPS:13101 and Action Manager:13104.
14. Choose to register the Administration and Monitoring Console as a system service. Note that this does not follow the normal Linux conventions, but adds a line to `/etc/inittab` instead.
15. Check the summary and click *Install*. The process takes about 4 minutes.

16. Choose **not** to start the configuration editor, as we do not want to run it as *root*. Click *Finish*.
17. To correct the Solution Directory bug, edit `/opt/IBM/TDI/V7.1/bin/defaultSolDir.sh` and change it to read:

```
TDI_SOLDIR="${HOME}/TDI"
```

    This will make the choice effective for all users, not just for *root*.
18. Find the TDI 7.1 fixpack and its associated README file in `~student/files` – read the instructions and install the fixpack.
19. Close the root shell.

# 3 The Config Editor

The TDI 7.1 Config Editor (CE) will not start unless the solution directory exists, so we must create it:

```
cd
mkdir TDI
```

Start the CE from the *Applications* menu.

Version 7.0 only installs menu items for the user that installed the software. One way around this is to start the CE from a command-line:

```
/opt/IBM/TDI/V7.0/ibmditk &
```

You now have to select a workspace. Accept the default, which puts the workspace under the Solution Directory. Tick the box to use this as the default, and click the button to start the Configuration Editor.

The next job is to create a project: Click the *Create Tivoli Directory Integrator Project* link. Call your project *TDIclass* and click *Finish.*

On most screens the fonts are rather small by default. If you need to adjust them, look at *Window->Preferences->General->Appearance* in the CE.

You can use the mouse to drag the divisions between the various panels around to make best use of screen area.

# 4 A simple assembly line

The object here is to create an assembly line to convert data from a CSV file to LDIF.

1. Create a directory to hold result files:

```
cd
mkdir results
```

2. In the Config Editor, expand your project and expand the *AssemblyLines* item.
3. Click *New Assembly Line* in the second row near the top of the window. Call the line *CSVtoLDIF*. When naming things in TDI it is always best to use names that would be valid Java variable names, so avoid spaces and punctuation.
4. An assembly line editor will start. Select the *Feed* section and click *Add Component*.
5. Select *Connectors* in the filter section to reduce the size of the list. Scroll to find the File System Connector and select that.
6. Change the name to *ReadCSV* and the mode to *Iterator.* Click *Next*.
7. Browse for the file or type its name:
   `/home/student/files/people-1.csv`
8. Click *Next*.
9. Select the CSV Parser.
10. Change the field separator to a comma and click *Finish.*
11. You now see the connector editor. You can review your settings by selecting the *Connection* and *Parser* tabs.
12. Select the *Input Map* tab. Click the *Connect* button and then *Next* to discover the column names and read a sample row of data.
13. Use shift-click to select all the attributes and then drag them left into the Work object.
14. Select the *Data Flow* part of the assembly line and add another connector. This will also be a File System Connector. Call it *WriteLDIF* and select *AddOnly* mode.
    The output file will be `/home/student/results/people-1.ldif` – This file does not exist yet so you will have to type its name.
15. Select the LDIF parser and accept its default settings.
16. In the Output Map tab, click *Add*. Select the four attributes that were read in from the CSV file and click *OK*.
17. You can now see a summary of the mappings by selecting *Data Flow* or by clicking the *Show Mapping* button.

18. Your assembly line is now ready to run, so save it using the floppy-disk icon and run it using the green *Run in Console* icon in the assembly-line editor.
19. A 'run editor' appears where the assembly-line editor was, and the log output from your assembly line is displayed in it. If all goes well, you should see that your two connectors each processed 6 entries.
20. Inspect the LDIF file that you have created. It will not load into an LDAP server at this stage as various things are missing, but it should be in the right format.

There is a problem in TDI v7.0 that causes long delays when starting assembly lines from the config editor. It is fixed in 7.1 but if you ever have to use 7.0 you should add this line to `/etc/hosts` as a workaround.

```
127.0.0.3 castor.exolab.org
```

# 5 TDI Debugger

The debugger is a very powerful TDI feature. Even in a simple assembly line it helps to see what is going on.

1. Select the tab where your *CSVtoLDIF* assembly line is being edited.
2. To the right of the green *Run* arrow is a button to start a debug session. Click it: a debug window will appear. This will initially show the *Data Stepper* view, which is a new feature in TDI 7.1.
3. Experiment with the *Next* button to see data flow through your assembly line. You can also use the →| button on the connector attribute panels to run the AL up to the point where the values are updated. Continue until the AL finishes.
4. Now try the full debugger by clicking the *Debugger* button.
5. Restart the AL using the green *Restart* button.
6. Drag the boundary between the *Data Stepper* pane and the *Watch List* pane to the left so that you can see the watch list properly.
7. Each item in your assembly line now has a check-box beside it: these are breakpoints where you can force execution to pause while you inspect data. Set the breakpoint for *WriteLDIF* and click the *Continue Execution* button. You will see that the work object is available in the data view and you can expand it to see the current attributes and values.
8. Click the *Continue Execution* button again and note the new values in *work*.
9. Hover the mouse cursor over each of the buttons in the debugger to see what they do. Try the *Step Into* and *Step Over* buttons. In this simple case they have much the same effect, but the difference will be useful later when we start using hooks.
10. Double-click the *WriteLdif* connector. A script-editor panel will open. Click the *Breakpoint Condition* button and add this to the script area:

```
work.surname == "Trott"
```

11. Click *Close* to recover the screen area used by the script editor, and re-run the AL: it will stop when it reaches the entry where the surname attribute has the value “Trott”. This is very useful where certain data patterns cause problems that need to be diagnosed in the debugger.
12. Close the debugger panel.

# 6 Attribute Maps

So far we have just read in some attributes from the CSV file and written them out to an LDIF file without change. The LDIF is not usable though: it lacks a valid DN and objectclass on each entry, and the attribute names do not fit the normal LDAP schema. We can use maps to correct this.

## 6.1 Using the output map

1. Select the AL Editor tab for your *CSVtoLDIF* assembly line.
2. Go to the output map of the *WriteLDIF* connector.
3. Click on the component attribute *surname* and change its name to *sn*
4. Change *username* to *uid* and *firstname* to *givenName* and *postcode* to *postalCode*
5. Add a new attribute to the output map, and call it *$dn* – this is a special name that the LDIF parser uses to generate the DN line in each entry.
6. Double-click on the Assignment part of your new attribute. A new editor pane will open. Replace the default mapping with this:

```
"uid=" + work.username + ",dc=people,dc=example,dc=org"
```

   This is a JavaScript expression that builds a valid DN from literal text and the value of the *username* attribute from the *work* object.
7. Close the expression editor, save the configuration and run the assembly line. Look at the LDIF file: much better but still not enough for an LDAP server to accept it. As a minimum we need to generate a valid *cn* (common name) attribute and an *objectClass* attribute – probably with several values.

## 6.2 In-line attribute maps

The new attributes will be useful in other connectors in the future, so we will create them in the main flow of the assembly line.

1. Start by adding a new Attribute Map component and calling it *LDAPAttrs*. Drag it to the top of the Data Flow section of the assembly line, above the *WriteLDIF* component.
2. Select the *LDAPAttrs* component and add a new attribute to it called *cn*. You will see that the default assignment is *work.cn* – double-click on this to open the JavaScript editor, and replace it with:

```
work.firstname + " " + work.surname
```

   Note that if you pause after typing “work.” you will be shown a scrolling list of valid methods and variables to select from.
3. Now add a new attribute called *objectClass*. This needs to be multi-valued. There are several ways to achieve this: we will simply return an array of fixed text strings, so replace the mapping with:

```
["inetOrgPerson", "organizationalPerson", "person"]
```

4. Several other attributes will also be of general use so we will move the mappings from the *WriteLDIF* component to the *LDAPAttrs* component. Click the *Show Mapping* button at the top of the assembly line editor pane to see the overall flow of data within the assembly line. Select the *$dn, givenName, postalCode, sn* and *uid* mappings in the *WriteLDIF* connector and type ctrl-X to cut them out. Now select the *LDAPAttrs* component and type ctrl-V to paste them in.
5. We still need output mappings in the *WriteLDIF* connector so select it and add the *objectClass*, *cn, sn, postalCode, uid* and *$dn* attributes to its output map. Accept the default mapping, which just copies them from the *work* object.
6. Save the configuration and run the assembly line in the full debugger. Step through it a few times while watching the work object to see the effect of the attribute-map component, and then let it run to completion.
7. Inspect the output file. Make sure that each entry has the correct attributes.

# 7 Flow Control

You have probably noticed that some entries in the CSV file do not have usable *firstname* and *surname* values. We will use a flow-control component to skip over these.

## 7.1 Using IF

1. Open the editor for the CSVtoLDIF assembly line, and select any component in the flow section.
2. Add an *IF* component. In its editor, make sure that *Match All* is ticked and then add 'Has Value' tests for both *firstname* and *surname*. When you add a new condition you can click in each column to edit the value.
3. When you click *Finish* you will be asked whether you want to add a component to the branch. Select *No*.
4. Drag the IF component to the top of the Flow section, and expand it. You will see a note about the branch being empty.
5. Drag the *LDAPAttrs* and *WriteLDIF* components into the branch so that they will only be executed if the work object contains both a firstname and a surname. If you cannot drag the components in, try creating a temporary component in the branch first.
6. Run the AL and inspect the output file. Has the branch worked? For a robust solution you may need more than one condition for each attribute.

## 7.2 ELSE

Silently skipping entries is not always a good idea, so we will add a log line to record what has happened.

1. Add an *ELSE* flow-control component to the end of the AL, and add a script component inside the new branch. The script should be based on the *Dump Work Entry* template, with this added at the top:

```
// Report skipped entry
task.logmsg("### Skipping entry as it needs both
   firstname and surname values");
```

2. Step through the AL in the debugger and note how the script is handled.
3. Inspect the output file and the execution log.

# 8 Database Lookup

In this exercise you will connect to a MySQL database and extract more information about the people listed in the CSV file.

## 8.1 Installing the MySQL JDBC connector

TDI does not come with drivers for all relational databases, so the first task is to install the MySQL JDBC connector.

1. Make a temporary directory and unpack the connector distribution in it. This is available from www.mysql.org and a copy is in the class kit: ~student/files/`mysql-connector-java-5.1.12.tar.gz`
2. Become root and create the directory: `/opt/IBM/TDI/V7.1/jars/local`
3. Copy the driver file `mysql-connector-java-5.1.12-bin.jar` into that directory.
4. Close the root shell.
5. You will need to restart the CE so that it detects the new connector library.

An alternative installation process that does not involve using *root* powers is to place the .jar file in any convenient directory and then add that directory to the CLASSPATH environment variable when running TDI.

## 8.2 Configuring the TDI connector

1. Add a JDBC Connector to your assembly line. Call it *DBLookup,* put it in Lookup mode and set these connection parameters:

| Parameter | Value |
|---|---|
| JDBC URL | jdbc:mysql:///classroom |
| JDBC Driver | com.mysql.jdbc.Driver |
| Username | student |
| Password | example |
| Table name | [see next step] |

2. Click the *Select* button beside the *Table Name* box. The CE will connect to the database and obtain a list of valid tables. Select 'person'. If the CE reports a connection problem you may need to save the config and restart the CE so that it finds the new connector library.
3. Move the new connector to be just above the *WriteLDIF* connector.
4. Go to the Input Map of your new connector and click *Connect.* Click *Next* a few times to see some typical data rows.
5. We want to get the phone number from this database, so drag *phone* from the schema into the work object and rename it to *telephoneNumber*.

6. We need to select the correct row to import data from. This is done using *Link Criteria*. Select the *Link Criteria* tab and add a new item to it. Hover the mouse cursor over each box to see the tool-tips. Choose to match *username* in the database against *username* in the work object, which is represented by *$username*
7. Add the new *telephoneNumber* attribute to the output map of the *WriteLDIF* connector.
8. Run the AL and inspect the output file. Make sure that the *telephoneNumber* attribute is included.

Note that TDI 7.0 up to Fix Pack 0004 has a bug which prevents the CE from connecting to databases once link criteria have been defined. It throws an 'Invalid thread access' error if you use the *Connect* button. The runtime system is not affected. [APAR IO12340]

# 9 Hooks and Scripts

In this exercise we will start to deal with an input file where there is no reliable join field. The file `people-2.csv` represents a more likely output from a personnel system, as it does not have a *username* field. The only join fields that we get are the *firstname* and *surname* so there will be ambiguities.

## 9.1 Preparation

1. Select your *CSVtoLDIF* assembly line in the left-hand (Navigator) panel, copy it and paste it back. Give it a new name: *HRtoLDIF*. Close the tabs relating to *CSVtoLDIF* to avoid confusion.
2. Open the new AL and change the *Connection* for the feed connector to use the `people-2.csv` file. In the input map, delete the old schema and mapping items and re-discover the schema. You will see that *username* has gone and *employee* has replaced it. Map all the schema attributes into the work object.
3. As this data source does not supply the username/uid attribute that later components depend on, we need to construct one. Add an item to the input map to build a *uid* attribute from the fixed test string “hr” followed by the employee number.
4. Delete the mapping for *uid* in the *LDAPAttrs* connector, and modify any mappings that used *username* so that they use *uid* instead.
5. Change the link criteria in the *DBLookup* connector to match using both the firstname and the lastname. Make sure the *Match Any* box is not ticked as we want both conditions to be satisfied.
6. Change the output filename in the *WriteLDIF* connector to be `people-2.ldif`
7. Run the assembly line. What happens?

## 9.2 Dealing with missing data

The AL threw an error because the database did not find a row matching one of the rows in the CSV file. In this case we do not care much, because we are only using the SQL data to add phone numbers to people's entries. Some people may not have phone numbers, so it is reasonable to ignore this error.

1. Open the *DBLookup* connector and go to the *Hooks* tab.
2. Look at the list of hooks. Hover the mouse cursor to get a little more information about them. Note that the list changes if you change the connector mode.
3. Select the *On No Match* hook so that it gets a tick in its box. In the script area on the right, add this JavaScript comment:

```
// Ignore no-match errors
```

The fact that the hook is enabled means that we have taken responsibility for the no-match condition so TDI will not throw an exception and stop the AL if no entry is found. The script does not have to do anything.

4. Run the AL. What happens?

## 9.3 Multiple hits

Another common problem in integration tasks is a work entry that matches more than one entry in a database. This can be more serious than a no-match condition, as it means that we probably have data but we don't know which result to use. The first thing to do is to report the problem and continue working, so that we can get a feel for the scale of the problem.

1. Select the *On Multiple Entries* hook in your *DBLookup* connector and insert this script:

```
// Warn of multiple hit
task.logmsg("#### WARNING: multiple hits in database
        for " + work.firstname + " " + work.surname);
```

2. Run the AL. What happens now?
3. Check the output file.
4. Run the AL in the debugger. Check the *Attributes* box so that input and output maps are displayed. Expand the maps. Step into each sub-component and watch what happens to the *conn* and *work* objects.

At any time you can check what would happen without a hook-script by un-ticking the box beside the hook name. The hook will be disabled but the script will stay in place to be re-enabled later.

## 9.4 Script objects and methods

TDI scripts are written in JavaScript, but most of the objects that they manipulate are from the underlying Java code. All such objects are described in the *JavaDocs* installed with TDI.

1. Iconise the config editor and open a file browser.
2. Browse to `/opt/IBM/TDI/V7.1/docs/api` and click on `index.html`
3. A web browser will open, showing an overview and navigation page. Find these items and scan the descriptions:

| | |
|---|---|
| UserFunctions | This describes the *system* object |
| RSInterface | Describes the *main* object |
| TaskInterface | Describes the *task* object |
| Entry | Describes *work*, *conn*, etc. |

4. Iconise the browser and return to the CE

# 10 Connector Loops

Connector loops have many uses. Here we will use one to do a lookup in a small XML file. The idea is to read each entry in the file until we find one where the employee ID matches the work object. This would be very inefficient in a large file.

1. Have a look at `~/files/cars.xml` to see what the data looks like.
2. In the *HRtoLDIF* AL, add a connector loop. Call it *LookupCarNumber*. Do not add a component to the loop yet.
3. Drag the loop component up above *WriteLDIF* and select it.
4. Make sure that the connector inherits from the basic File System Connector (this should be the default).
5. In the *Connection* tab, select the input XML file.
6. In the *Parser* tab, select the basic XML parser.
7. Each entry in the XML file is wrapped in 'Car' tags like this:

```
<Car>
        <carReg>LDA273F</carReg>
        <employee>3782</employee>
</Car>
```

   Set the parser's *Entry Tag* to be 'Car' to match this format. (The default '*' will work too, but specifying the tag protects against problems caused by any non-Car records in the file).
8. Go to the input map and read a few entries to check the settings. Copy all the attributes into the work entry.
9. Expand the loop and double-click on the empty-branch marker to add a component.
10. Add an IF component, and call it *IfFoundEmployee.* Leave the match conditions blank and do not add a component to the branch yet.
11. We want to match the *employee* attribute from the XML file against the *employee* attribute in the work object. This is a problem as both have the same name so the XML data overrides the earlier data while inside the loop. Return to the connector loop's input map and change the names of the attributes to *XMLCarReg* and *XMLEmployee* to avoid this clash.
12. In the IF component, set the condition to match *XMLEmployee* against *$employee*
13. Add an attribute map component to the IF branch and use it to map *XMLCarReg* into *carLicense* (note the American spelling).
14. Add a script component to the branch after the attribute map. Select the *Exit Branch* script and call it *StopLoop*.
15. The script template has a call to *system.exitBranch()* - find this in the JavaDocs and choose an appropriate parameter to cause it to terminate the connector loop.

16. Go to the output map of the *WriteLDIF* connector and map *carLicense*.
17. Save the AL and step through it in the debugger to make sure it does what you expect. Check the output file.

# 11 Adding and Updating LDAP entries

It is time to connect to another database: an LDAP server this time. This represents the central enterprise directory of the organisation, and we want to enrich the data held there. Some people already have entries, as this service is used by corporate Unix and Web servers for authentication and authorisation.

## 11.1 Building the LDAP Server

Some pre-configured LDAP servers are included in the class kit. They are all based on OpenLDAP, and can be started, stopped, and reloaded from the command line. These servers do not run as root.

1. Open a shell window and type these commands:

```
cd ~/ldap/simple
x-rebuild
```

2. The LDAP server is now running. It has these characteristics:

| | |
|---|---|
| Hostname | localhost |
| Port | 1389 |
| Management DN | cn=root,dc=example,dc=org |
| Management password | example |

3. Check that the server has loaded properly:

```
x-anon-search
```

   You should get all the public-view data from the server in LDIF.

Note that the 'x-' commands all depend on the working directory to identify the LDAP server to operate on. You will find a complete set of handy commands in `~/ldap/bin`

You can re-run the commands in step 1 above at any time to clear out and reload the LDAP server.

## 11.2 Installing Apache Directory Studio

Command-line tools are very good for getting things done quickly, but are not so good for browsing data. We will install Apache Directory Studio for this purpose.

1. Find the distribution file in ~/files and unpack it in your home directory. You will get a directory called something like `ApacheDirectoryStudio-linux-x86-1.5.2.v20091211`
2. Go to that directory and run the binary:

```
cd ApacheDirectoryStudio*
./ApacheDirectoryStudio
```

3. Studio complains that it cannot find a Java binary. To fix this, create a text file called `ApacheDirectoryStudio.ini` in the same directory containing these two lines:

```
-vm
/opt/IBM/TDI/V7.1/jvm/jre/bin/javaw
```

4. Re-run the ApacheDirectoryStudio binary: it will now use the IBM JVM that was installed with TDI. In fact, both TDI and Apache Directory Studio are based on Eclipse so it should be possible to run both in the same Eclipse instance. We will not try that today.
5. Select the Workbench, and a familiar Eclipse screen will appear.
6. In the *Connections* panel (bottom left) click on the yellow LDAP icon to create a new connection. Configure the connection using the LDAP server data given above. Check that you can browse the LDAP data.

## 11.3 Connecting TDI to LDAP

TDI has a rich set of LDAP facilities. We will use the LDAP client connector.

1. Add an LDAP Connector after the *WriteLDIF* connector. Call it *CorpLDAP* and put it in Update mode.
2. Configure the basic connection parameters, then click the *Contexts* button to get the list of suffixes that the server supports. Select `dc=example,dc=org`
   If you get a null pointer exception at this point you will have to enter the context by hand. The problem is that TDI 7.1 is not explicitly requesting the *namingContexts* attribute: as this is an operational attribute it does not get included in the results by default (fixed in FP0002).
3. Use the data browser on the Output Map tab to check the connection.
4. Close the data browser connection, and delete all attributes from the schema panel to avoid confusion in the next step.
5. Add the following attributes to the output map:
   `$dn carLicense cn givenName objectClass postalCode sn telephoneNumber uid`
   (If you had left the discovered attributes alone in the previous step you would have to scroll more times to do this).
6. Set the Link Criteria to match on *cn* (common name).
7. Save the config and run the AL. It fails, so read the first few lines of the error report.
8. The error report complains about an undefined operation code, which is not very helpful, but it does mention the attribute concerned: *carLicense* which came from the XML parsing connector. Go to the input map of the *LookupCarNumber* connector and scroll the schema display horizontally to see all the data. Compare this with the schema for the *ReadCSV* connector: you will see that they return objects with different Java classes. This has not mattered so far, but the LDAP connector is capable of making use of extra information in AttributeValue objects, and the XML parser has supplied such an object without the extra fields.

9. The simple solution is to convert the offending attributes to strings, so go to the input map of the *LookupCarNumber* connector. Double-click on the *Assignment* field for each attribute in turn and add `'.getValue()'` to the end of each expression:

```
conn.carReg.getValue()
conn.employee.getValue()
```

   This causes the expressions to return the first (and only) value as a `java.lang.String` object.

10. Save the AL and run it again. A different error will stop the AL. This time it is a schema violation, complaining that *uidNumber* is not allowed in the entry. This seems odd, as we have not created such an attribute in the AL. The DN of the offending entry is given, so have a look at it in Apache Directory Studio.

11. You will find that this is the 'Caroline Dobson' entry, which was in the LDAP store before the AL was run. It has been renamed as the LDAP connector supplies a new DN, but it still has the original data which includes a *uidNumber* attribute.
    There are really two problems here: we are renaming the entry for no good reason, and we are forcing in an inadequate set of new object classes.

12. Go to the output map of the *CorpLDAP* connector. You will see two columns headed *Add* and *Mod.* These determine whether the attribute takes part in add and modify operations respectively. We do not want to change the DN and *uid* of existing entries, so click on the *true* value in the *Mod* column for *$dn, objectClass* and *uid*. The value changes to false, so now these attributes will not be used in modify operations. We still need them when adding new entries so the values in the *Add* column should remain *true*.

13. This experiment has messed up the 'Caroline Dobson' entry, so rebuild the LDAP server to its starting position:

```
cd ~/ldap/simple
x-rebuild
```

14. Save and run the AL in the debugger. You will recognise the error this time, so create a script to log the problem and continue.

15. Save and run again. Yet another error! TDI is trying to create an entry that already exists, but we know that it only adds new entries if it does not find an existing entry to modify. Can you see what is going on?

16. Add a script in the *CorpLDAP* connector's *Update Error* hook. It should log a suitable warning message and dump the *error* and *work* objects.

17. Save the AL, rebuild the LDAP server, and run the AL again. It should work this time, and the execution log should contain your warning messages: these should help you to diagnose the problems in the data.

## 11.4 Summary

This exercise has hit one problem after another. It may seem tedious, but integration tasks are often like this: what we are really doing is gradually learning about the data and devising strategies to work around its imperfections. In a real-life situation it could take many days, thousands of log entries, and many discussions with data owners to evolve the best approach.

Part of the problem with the Add operations is due to the way this AL generates *uid* values and then uses them as part of the entry DN. Using user-visible attributes in DNs often leads to trouble, which is why the pre-loaded LDAP data uses random *uniqueIdentifier* values as RDNs for people entries. See my paper on LDAP Schema Design for more about this:

http://www.skills-1st.co.uk/papers/ldap-schema-design-feb-2005/index.html

# 12 Property Stores

Properties allow you to factor out things like connection data, addresses, and passwords from the main configuration file.

TDI supports multiple property stores. At the least you will find global Java properties and per-project properties. You can create more stores if you need them.

We will use a property store to hold the LDAP connection data.

1. In the *Navigator* panel, expand *Resources* and then *Properties*.
2. Double-click on the property store whose name matches your project name. A property editor will open.
3. Click *Add Property* and call the new property *LDAP_URL*
4. In the property list, click in the *Local Value* field for *LDAP_URL*. Type this value and press *enter*:

   ```
   ldap://localhost:1389/
   ```

   Note that you cannot directly edit the *Server Value* field. To update that you must click the *Send properties to Server* button.
5. Create each of the properties in this table:

   | Name | Value |
   |---|---|
   | LDAP_URL | ldap://localhost:1389/ |
   | LDAP_MGR_DN | cn=root,dc=example,dc=org |
   | LDAP_MGR_PASS | example |
   | LDAP_SEARCH_BASE | dc=example,dc=org |

   There is nothing magical about the names: you can make them up to suit your system – but they should be valid variable names.
6. Click the floppy disk icon to save the contents of the property editor. This makes the values available to other parts of the configuration editor.
7. Click the *Send properties to Server* button to make the values available to the runtime process.
8. Open your *HRtoLDIF* AL and go to the connection tab of the *CorpLDAP* connector.
9. Click on the "?" button by the *LDAP URL* field (clicking on the field label also works, as this was the convention in previous versions of TDI).
10. Select *Use Property* and then select the *LDAP_URL* property from the scrolling list. You may need to make the window larger to get the list to scroll properly. Note the name `TDIclass:LDAP_URL` as this could be typed directly. It can also be used in scripts.
11. Replace the other config parameters with properties. Note the '(property)' tags that appear in the field labels.

12. Save your config and test it.
13. Return to the Property Editor and look at the *Connector* tab. The Collection Path is a filename: inspect the file. In a production system you could replace this file with one containing connection data for the production LDAP server.
    NOTE: the default permissions on this file allow all users to read it. As property files are likely to contain sensitive data you should restrict access to them in production environments.

# 13 Handling multiple-hit results

Finding multiple entries in response to a lookup is not always an error.

The MySQL database has a table called *phone* which maps usernames to phone numbers. Some people have one number, some have none, and some have several.

1. Build a new AL that Iterates through the file people-1.csv and writes an LDIF file.
2. Now add a Connector Loop to do SQL lookups. Set it to iterate over all database rows where the username matches the username in the work object.
3. Add each phone number to the work object's *telephoneNumber* attribute. You may find this code fragment useful:

```
if (! work.telephoneNumber) {
     work.newAttribute("telephoneNumber");
  }
work.telephoneNumber.addValue(work.getString("phone"));
```

4. Inspect the output file and compare it with the input CSV file. Look in the file `phone-table.csv` to see what the MySQL phone table contains.

# 14 Server-Mode connectors

Use the HTTP server-mode connector to receive queries from the net on port 8088. Do **not** add a parser, as this connector already understands the HTTP format.

Find out what happens when you use Firefox to read URLs starting http://localhost:8088/

Choose a URL format suitable for simple LDAP lookups – perhaps something like:

```
http://localhost:8088/sn/Smith
```

Use the incoming query string to do a search of the form 'sn=...'.

Return a formatted result.

Test with Firefox.

# 15 Delta-Mode connectors

Delta mode is really useful for dealing with data sources that do not supply change info. It works by storing a copy of each entry in the TDI system database, so it may not be suitable for really big datasets.

## 15.1 Configure the System Store

TDI uses the *System Store* to hold delta information. The default configuration stores the data in the software installation directory: this is only writeable by *root* and in any case it is not wise to mix code and data so we must set a more suitable location.

1. Use the Navigator panel to open
   *TDIClass -> Resources -> Properties -> Java-Properties*
2. Read the properties from the server and find *com.ibm.di.store.database*
3. The value of this property is a URL which contains an embedded pathname. Change `/opt/IBM/TDI/V7.1/TDISysStore` to `/TDISysStore` leaving the rest of the URL intact:

   ```
   jdbc:derby://localhost:1527/TDISysStore;create=true
   ```
4. Send the updated property to the server.
5. Save the properties.

## 15.2 Connect the CE to the System Store

1. The last item in the main *Navigator* panel for the *TDIClass* project is *Solution Logging and Settings*. Double-click to bring up its editor.
2. Select the *System Store* tab. This is where we configure the connection between the CE and the System Store. It has no effect on the runtime environment.
3. At the end of the heading *Config Instance System Store* is a small triangular icon. Click on this to get a menu and select *Derby Networked*.
4. Make sure that the Database URL matches the one that you edited above.
5. Click *Start* to start the System Store. Wait for the acknowledgement window.
6. Click *Test Connection* and wait for the result.
7. Save the configuration.

## 15.3 Simple Delta Mode

1. Take a copy of one of the CSV files. Build an AL with a filesystem connector to iterate through the file and dump each entry to the execution log. Test it.
2. What happens when you run the AL a second time?

3. Now go to the *Delta* tab of the filesystem connector and enable Delta Mode. Choose a suitable unique attribute name and give a name for the Delta Store.
4. Run the AL.
5. Run it again. What happens this time?
6. Modify one entry in the CSV file and try again.
7. Go back to the Delta tab and delete the Delta Store. Re-run the AL to see the effect.

## 15.4 Using Delta Data

When an entry is changed, the Delta mode algorithm puts a note of the actual changes into the work object. This can be used to do more intelligent updates in some connectors. The LDAP connector is one of those.

Add an LDAP connector to your AL and experiment with its Delta mode.

# 16 TLS and X.509 certificates

Many TDI connectors have the ability to use SSL encryption on network connections, but this has been deprecated by the IETF standards for some years. The preferred security mechanism is TLS (Transport Layer Security) as defined in RFC2246, RFC4346 and RFC5246. Unfortunately, TDI 7.1 does not yet have a simple tick-box to support TLS in connectors so we need to add some code (not very much though).

## 16.1 TLS in LDAP

The *simple* LDAP server used in earlier exercises has support for TLS.

1. Start the server and test it:

```
cd ~/ldap/simple
x-start-ldap
x-anon-search sn=trott
```

2. You should see one entry. Now do the same search with TLS:

```
x-anon-search -ZZ sn=trott
```

3. You should see the same entry, but this time the *exampleAccountStatus* attribute is visible. This is because the access-control rules have been set to only disclose that attribute to normal users if the connection has at least 56 bits of cryptographic protection
(see `~/ldap/simple/openldap/slapd.conf.acls` for details of how this is done)

4. Start a root shell and inspect the log file for the server:

```
less +G /var/log/localmessages
```

you should be able to see the TLS negotiation and its result.

The X.509 keys and certificates are in `~/ldap/x509` along with a note on how they were generated.

## 16.2 A new assembly line

Make a new AL for the TLS tests.

Put an LDAP iterator in the feed section and set it to read all the entries that match objectclass=person. Use these credentials:

Username: `uid=u1,dc=people,dc=example,dc=org`

Password: `secret`

Put a file system connector in the flow section and set it to write the entries to a new LDIF file.

In both connectors use '*' in the maps so that all attributes are copied to and from the work entry.

Test your AL: you should see 10 entries, but none with the *exampleAccountStatus* attribute.

## 16.3 Negotiating TLS

The whole point of using TLS rather than SSL is that TLS can be negotiated after the application protocol has started. Thus an LDAP server for example only needs to listen on one TCP port to handle both encrypted and unprotected sessions.

From a TDI point of view, this means that we can intervene in a connector hook to add TLS protection and the rest of the system does not need to know what we have done. Here is some code to start TLS:

```
try{
        var ctx = thisConnector.connector.getLdapContext();
        tls = ctx.extendedOperation(new
            Packages.javax.naming.ldap.StartTlsRequest());

        task.logmsg("Negotiating TLS");
        tls.negotiate();
        task.logmsg("TLS OK");
} catch (E) {

        task.logmsg("TLS negotiation FAILED");
        task.logmsg(E);
        tls.close();
        tls=null;
        system.abortAssemblyLine("TLS negotiation FAILED")
}
```

Drop the code fragment into the *After Initialise* hook of your LDAP connector and try it (there is a copy in `~/files/TLS-code-fragment` which you can paste in).

Re-run your AL: you should get a failure message even with a TLS-enabled LDAP server, as the cryptographic trust relationship must be set up before TLS negotiation will succeed.

## 16.4 Key and Certificate management

To set up the cryptographic trust relationship you need to import the CA certificate that signed the LDAP server's certificate. (If you are using a self-signed certificate then just import that, but such relationships are harder to manage in the long run).

The complication here is that TDI already uses SSL/TLS when talking to the runtime server, so you must preserve or replace the keys used for that purpose. By default, the same file is used as both keystore and truststore. To find its name and password:

1. Use the Navigator panel to open
   *TDIClass -> Resources -> Properties -> Java-Properties*
2. Read the properties from the server and find the values for:
   *javax.net.ssl.keyStore*
   *javax.net.ssl.keyStorePassword*
   *javax.net.ssl.trustStore*
   *javax.net.ssl.trustStorePassword*

The filenames are relative to the solution directory, so you should find that both keystore and truststore are in `/home/student/TDI/serverapi/testadmin.jks` and the password for it is `administrator`

3. Note that the admin certificate in the distributed file expires on 7th September 2010, so if your CE stops talking to the development server on that day you will know why!
4. Start the IBM Key Manager by clicking the 'key manager' button at the top of the CE window. Use it to open the keystore that you located above.
5. Select 'Signer Certificates' and add the LDAP Certification Authority from the file `~/ldap/x509/ca.cert` - TDI should now be able to use TLS when making connections to any server whose certificate was signed by that CA.
6. Restart the CE and check that it can talk to the development server before continuing. If it fails, you will have to kill the development server manually before trying again (look for a Java process running something under `/opt/IBM/TDI`).
7. Re-run the AL: you should see that TLS negotiation succeeds.
8. Check the output file: are the *exampleAccountStatus* attributes visible? Look at the LDAP server log in `/var/log/localmessages` – can you see what has gone wrong?
9. Move the TLS code to a more suitable place (note that this particular problem only affects some connector modes: *iterator* is one of them).
10. Re-run the AL and check that the *exampleAccountStatus* attribute is now visible in the output file.
11. Look at the LDAP server log again. There is still a security problem: can you see it? To fix this one we would have to move the LDAP authentication step into the script rather than using TDI's built-in code.

## 16.5 A malicious LDAP server

TLS can provide protection against rogue servers and connection hijacking.

1. Stop the *simple* server

```
cd ~/ldap/simple
x-stop-ldap
```

2. Start the attacker's server

```
cd ~/ldap/imposter
x-rebuild
```

3. This server has an identical configuration, but it uses different certificates. The CA and server certificates were made using exactly the same commands and parameters as the 'real' ones, but the keys are different.
4. Run your AL again. What happens?

# 17 Preparing for deployment

You will not want to run assembly lines from the config editor when the system goes into production service. In many cases the ALs must run on servers where the CE is not even installed. To make this possible you need to extract the information that the runtime system needs.

1. Create a directory called `export` to hold the production-ready files.
2. Right-click on the project name in the navigator panel and select the *Export files* option.
3. Select your new directory and click OK.
4. Choose to run just the CSVtoLDIF assembly line at startup.
5. Click OK.

Inspect the files that the CE has written into your export directory. These are the minimum set that you need to copy to the production machine. The file *TDIclass* is a script to start the runtime and the selected ALs: try it, and check that the output file is updated in `~student/results`

# 18 AMC

The Admin and Monitoring Console is a web app that allows you to control certain TDI functions from a browser. It is described in the Administration Guide.

At install time we asked for AMC to be registered as a system service. Unfortunately the installer gets this wrong on Linux. It puts a startup clause in `/etc/inittab` (which is OK but not normal) but the command used is wrong. To compound the problem, this version of AMC writes logfiles into the software installation area so it cannot be run by a non-root user.

1. Start a shell window and become *root*.
2. Run this command:

   ```
   /opt/IBM/TDI/V7.1/bin/amc/amcservice.sh start amc
   ```

3. The command will issue a couple of messages and sit waiting for something to happen. It does not return to the shell prompt.
4. Start Firefox and go to this URL:

   ```
   http://localhost:13100/ibm/console
   ```

   Note that the URL given in the version 7.0 Admin Guide is wrong. You will be redirected to an HTTPS port where the SSL certificate is incorrect: accept it for now.
5. Login to AMC. The username is *root* and the password is the Linux *root* password! (*example*)
6. Explore AMC. Refer to the Admin Guide for configuration info. You will need to create a Solution View before you can see any details about your server.

# 19 Data Cleanup and Reporting

An important early stage in any synchronisation project is to find out how far out of sync the existing data-sources are. TDI can help with this.

In the directory `~/files/data-cleaning-exercise` you will find files called `names.ldif` and `names.passwd`. The first one is a dump from a corporate LDAP service and the second one is a Unix password file that is supposed to match. Use TDI to report on the differences between the data sources.

If you want to load the LDIF into an LDAP server, copy the server in `~/ldap/skel` and modify it as necessary. It listens on port 9389.

If you want a relational database you can add tables to MySQL. See `~/files/mysql-setup` for some handy commands and examples.

# 20 LDAP to SQL synchronisation

In exercise 13 *Handling multiple-hit results*, you extracted phone numbers from an SQL table and generated LDAP entries. In this exercise you will complete the loop by synchronising data from LDAP to SQL.

1. Find the assembly line that you built in exercise 13 and add an LDAP adaptor to update the *simple* LDAP server. Run the AL and check that the correct phone numbers are inserted.
2. Now build an assembly line to synchronise from LDAP back into the *phone* table.
3. Test both ALs carefully to make sure that they work correctly. Here are some test cases to try:

   • Change the phone number in an LDAP entry with a single number already present.

   • Change the phone number in an LDAP entry with multiple numbers already present.

   • Delete all numbers from an entry.

   • Add a number to an entry that does not have one.

   • Add a new number to an entry that already has at least one.

# 21 Mailing List from LDAP Group

LDAP groups are defined as entries with *member* attributes. Each value of the attribute is the DN of an entry.

To use a group as a mailing list it is necessary to iterate through the *member* values and read the *mail* attribute from each one.

1. The LDAP server in `~/ldap/school-isp` has data representing a mail service provider for a group of schools. Load it like this:

```
cd ~/ldap/school-isp
x-rebuild
```

2. This server listens on port 3389. The other connection parameters are the same as the *simple* server.
3. This entry is a group with several members:

```
uniqueIdentifier=staff,ou=groups,uniqueIdentifier=0001,dc=mag
ic,dc=example,dc=com
```

4. Create a new AL. Add an LDAP connector to read the member attribute from the group.
5. You will need an attribute-value loop and another LDAP connector to read the individual member entries. Try to re-use the first LDAP connector if possible (right-click on it and select *Re-use Connector*).
6. Write a file containing one mail address per line. You may find the *Line Reader* parser useful.
7. Test the AL: it should write three mail addresses to the file.
8. Modify the AL so that the first LDAP connector accepts the DN of the group entry as an attribute in the work object. Add a *Form Entry* connector to supply some test data in the feed section of the AL.

# 22 Generating unique IDs safely using LDAP

When creating accounts it is often necessary to generate a unique identifier – a username, Unix UID, etc. Depending on the environment there are several approaches possible, for example:

- Use the date and time plus a running count:
  20100609104327.1234
  The count is added because system clocks may not have enough resolution to be sure that every ID is different – particularly on Windows.
- Use a true random number. If the number is large enough (e.g. 64 bits) and it is truly random then the risk of a clash is negligible. Be aware that a 32-bit number is usually *not* large enough! For the background to this, look up the “Birthday Problem” on Wikipedia.
- Use a hash derived from date and time plus a running count. This has the advantage of not exposing the date when a given ID was created. Use MD5 or SHA1 as the hash. Do not truncate the hash without first considering the Birthday Problem...
- Keep the next ID value to be used in a database or LDAP entry. This allows for much smaller ID values, but does require a safe atomic update.

LDAP does not normally provide transaction support so there is no general atomic update feature. However there is one case where an atomic update is possible: when a single entry is updated in a single LDAP operation then all parts of the update will succeed or they will all fail. This allows us to build a safe ID generator.

Imagine an LDAP entry containing a *uid* attribute with a numeric value. The algorithm to safely increment the value is:

```
REPEAT
     Read the uid from the entry: call the value N
     Update the entry:
          Remove uid=N
          Add uid=(N+1)
UNTIL (update succeeds)
```

Implement this as a TDI assembly line and test it. Does TDI issue the safe LDAP operation or does it simply replace the *uid* attribute with the new value? You can find out by single-stepping two copies of the AL so that both do the read before either does the update. The second AL to try the update should fail and go around the loop again. If it does not, you will have to resort to scripting the LDAP update operation to force the use of the safe method.

Other ways to do the same thing include the use of the 'unique attribute values' feature found on many LDAP servers. Be aware that LDAP servers running in multi-master configurations cannot usually guarantee to maintain unique attributes or atomic attribute updates of any sort.

# 23 Account Provisioning using LDAP

There are usually several stages to any provisioning process. It is important to keep track of progress in such a way that the process can be restarted at any point without breaking. In systems with a core LDAP directory it can be convenient to keep the provisioning status in the entry relating to the user.

In this exercise you will build a provisioning process for a schools ISP.

1. Start the `school-isp` LDAP server and browse its contents. See how each school has the same structure. Look at the attributes in the school entry and in the user entry.
2. Create two new tables in the MySQL database. These will represent the mail server and the web server, both of which need account data for each user:

```
mysql -u root -pexample classroom
create table mailsys (
        uid varchar(16),
        mail varchar(40),
        primary key (uid)
);
create table webservice (
        uid varchar(16),
        cn varchar(60),
        primary key (uid)
);
```

3. To save time we will misuse the *employeeType* attribute to hold provisioning state. In real life we would define a new attribute for this purpose. This attribute will hold data in the form 'key=value' to keep track of progress.
4. Build an AL that finds all entries where *employeeType* holds the value 'new=yes'. For each entry found it should replace that value with two new values: 'mail=required' and 'web=required'
5. Build an AL that finds all entries where *employeeType* holds the value 'mail=required'. For each entry it should create a row in the *mailsys* table holding the *uid* and *mail* values from the entry. It should then remove the 'mail=required' value and replace it with 'mail=provided'.
6. Build an AL that does a similar job for the *webservice* table.
7. Test your system by setting *employeeType* to 'new=yes' in an existing entry, then running each of your assembly lines. Run them again to make sure they do not repeat any work. Check that the *employeeType* attribute ends up with just 'mail=provided' and 'web=provided' values. Are there any cases where problems would be caused by two ALs running at the same time?
8. Running each AL by hand is very clumsy, so build a controlling AL that invokes each worker AL in turn. In a production system this might run

every few minutes or it might be triggered by a change notification from the LDAP server.

9. Consider how you would implement de-provisioning in this system. Build the necessary ALs. Make sure that it is possible (and safe) to de-provision a partly-provisioned account, bearing in mind that one or more of the services being provisioned might be down when the request comes in.

This approach to storing provisioning state data in LDAP depends on an atomic LDAP update to be safe. Implementing this in TDI requires scripting, as the normal LDAP update mode does not implement the attribute changes in a way that is safe for this purpose. This scheme also requires that the LDAP infrastructure implements safe atomic updates, which will not be possible in most multi-master server configurations.

If you cannot use LDAP to store the provisioning state, the same approach would work using an SQL database. The key is to make sure that the update transactions are atomic and that they only affect the relevant values.

# 24 Parsing complex log files

Processes such as webservers generate log files with long complex lines. We sometimes need to extract data from these, but TDI does not have a 'webserver logfile' parser. This problem can be solved using the script parser.

1. Create a new AL called ProcessWebLog.
2. Add a filesystem connector in iterator mode, and set it to read the file `web-log-sample`
3. Add a Script parser to the connector. In the parser, click the *Edit Script* button and look at the template that has been supplied.
   You should find three methods (functions): *writeEntry*, *readEntry*, and *querySchema*. *writeEntry* is only used in output mode: we are interested in the other two.
4. Go to the input map and connect to the data source. You should see one attribute appear in the schema: *line* – this is the only one mentioned in the *querySchema* method.
5. Click the *Next* button: you should see a new line from the logfile in the *line* attribute each time. This is not very useful as a parser, but it does show that the connector is working.
6. Edit the parser script again, and change the *querySchema* method to look like this:

```
function querySchema ()
{
   var attrs = ["line","domain","ip"];
   for (var i=0; i< attrs.length; i++) {
      var e = new com.ibm.di.entry.Entry();
      e.addAttributeValue("name",attrs[i]);
      e.addAttributeValue("syntax","String");
      list.add(e);
   }
   result.setStatus (1);
}
```

   The idea is to create several Entry objects, each describing an attribute for the schema. All the attributes will be strings.
7. Return to the input map, close the connection and re-connect. You should see new schema attributes *domain* and *ip*.
8. Now we need to start parsing the actual log lines. Regular Expressions provide a powerful text-matching facility that will help.
   Look at the *readEntry* method. You will see that it reads a line of text into *str* and then does some tests before calling *entry.setAttribute* to put the line into the *line* attribute in the *entry* object. The *Script Parser* chapter in the TDI Reference Guide describes how the parser communicates with the rest of the system.
   Add this after the line that says `counter++;`

```
var re = new RegExp("^([^ ]+) ([^ ]+) ");
var attrs = re.exec(str);

entry.setAttribute ("domain", attrs[1]);
entry.setAttribute ("ip", attrs[2]);
```

This does a lot of work in a few lines, so it is worth looking at it carefully.

The first line defines a regular expression that parses the first few items on the line. It saves some of the matched text into variables that can be used later. A rough description of the expression is:
*Starting with the first character of the line, match at least one non-space character and save the matched characters.*
*Match exactly one space.*
*Match at least one non-space character and save the matched characters.*
*Match exactly one space.*
*Ignore the rest of the line.*
For a full description of how regular expressions work, see the JavaScript Core Reference. Most books covering Perl or Unix command-line utilities will also have sections on regular expressions.

The second line executes the expression against the input line (*str*). If the pattern matches, the saved text is copied into elements of the array *attrs*. In this case there will be two elements because there are two 'saves'. The elements we want are indexed 1 and 2, as the whole string is copied into element 0 of the array.

We now have the first two element of the log line in the *attrs* array. The last two lines of the code fragment copy this data into the *domain* and *ip* attributes in *entry*.

9. Save the script and return to the input map. Close and re-connect, then click the *Next* button to see what happens. You should see data from the logfile appearing in the Sample Value column.
10. Extend the parser to extract the date field as well.
11. Add a DumpEntry script component to your assembly line, and run it to check that you get *domain, ip,* and *date* in the results.

## 24.1 Handling active logfiles

Logfiles on an active system grow continuously, and it is often useful to analyse them in real time. This means that we need to read all the lines in the file and then wait for more to be appended, rather like the Unix `tail -f` command.

1. Go to the Connection tab of your ReadLogFile connector. Expand the Advanced section at the bottom.
2. Click on the question-mark beside the Timeout box and read the description. Set a timeout of 0 (zero) so that the connector will wait forever for new lines to appear in the file.

3. Change the file path to `/home/student/files/log`
4. Now we need to simulate a webserver logfile. The script `dripfeed` is provided for this purpose:

```
cd ~/files
dripfeed web-log-sample > log
```

   The effect of this is to copy one line from the sample file to *log* every two seconds. You can see what is going on by running this in another window:

```
tail -f ~/files/log
```

5. Run your assembly line. You should see some entries reported very quickly, then the rate will slow down to one every two seconds.
6. In a practical system, instead of just dumping the work object we would probably analyse the entries and keep running counts. Another assembly line could make the data available via HTTP or SNMP.